Infinite Unlimited Churn

Dianne Foreback, Mikhail Nesterenko, Sébastien Tixeuil

Infinite Unlimited Churn

Dianne Foreback[1], Mikhail Nesterenko[1], and Sébastien Tixeuil[2]

[1] Kent State University, Kent, OH, USA
[2] UPMC Sorbonne Universités & IUF, Paris, France

Abstract. We study unlimited infinite churn in peer-to-peer overlay networks. Under this churn, arbitrary many peers may concurrently request to join or leave the overlay network; moreover these requests may never stop coming. We prove that unlimited adversarial churn, where processes may just exit the overlay network, is unsolvable. We focus on cooperative churn where exiting processes participate in the churn handling algorithm. We define the problem of unlimited infinite churn in this setting. We distinguish the fair version of the problem, where each request is eventually satisfied, from the unfair version that just guarantees progress. We focus on local solutions to the problem, and prove that a local solution to the Fair Infinite Unlimited Churn is impossible. We then present and prove correct an algorithm $\mathcal{UIUC}$ that solves the Unfair Infinite Unlimited Churn Problem for a linearized peer-to-peer overlay network. We extend this solution to skip lists and skip graphs.

1 Introduction

In a peer-to-peer overlay network, each member maintains the identifiers of its overlay neighbors in its memory while leaving message routing to the underlay. A peer-to-peer overlay network is inherently decentralized and scales up easily. Peer-to-peer overlays are well suited for distributed content storage and delivery. Recent applications of such overlays to internet telephony [10] and digital cryptocurrencies [28] further enhance interest in scientific studies of the principles of peer-to-peer overlay networks.

Due to the lack of central authority and the volunteer nature of overlay network participation, *churn*, or joining and leaving of peers, is a particularly vexing problem affecting peer-to-peer overlay networks. Churn may be cooperative, if departing processes execute a prescribed departure algorithm; or adversarial, if they just exit.

Infinite and unlimited churn. Every peer-to-peer overlay network has to handle churn. Usually, while the topological changes in the overlay required by the churn requests occur, the primary services of the overlay (*e.g.* content retrieval) are either considered suspended or they are disregarded altogether. In other words, the churn is considered finite, and the overlay network users just wait until join/leave requests stop coming, then the overlay network recovers and restores services. If churn happens again, the service suspension repeats. This may be tolerable if churn is infrequent since the overlay network is available most of the time. However, at the scales that peer-to-peer overlay networks achieve, peers nearly always wish to join or leave the overlay network. In this case, the service degradation caused by intermittent suspensions may become unacceptable. That is, it is necessary to consider *infinite churn* under which the overlay network has maintain services while handling it.

One way to handle churn is to engineer sufficient redundancy in the overlay network topology, so that if the peers leave, there are enough alternative paths to enable service

operation. In this approach, the amount of redundancy necessarily places a limit on the number of processes that churn concurrently: the churning processes must not sever all redundant paths. However, under heavy churn load and large scale, a peer-to-peer overlay network may breach this limit and collapse (that is, partition itself). Also, the necessity to add redundancy leads to wasted resources. Hence, there is an interest in studying the possibility of *unlimited churn*, where this is no bound on the number of concurrently joining or leaving processes.

This paper is an attempt to define and study infinite unlimited churn in peer-to-peer overlay networks.

Unfair and local churn. A request to join and, in cooperative churn, leave the overlay network is submitted to the overlay by the churning process. A churn handling algorithm is *fair* if it eventually satisfies every request. By contrast, a churn algorithm that allows the possibility, under infinite churn, to bypass indefinitely some requests (still guaranteeing progress, that is, satisfying some churn requests indefinitely), is *unfair*. Unfair algorithms are possibly more efficient.

Potentially, a churn handling solution may be straightforward and *global*: have some distinguished process manage all churn requests and process them. However, such a centralized global solution is not practical for large scale overlay networks as it creates a performance bottleneck and a single point of failure. In contrast, in a *local* solution, only processes in the immediate vicinity of the churning process are involved in processing the request.

In this paper, we study fairness and locality of churn solutions.

Our contribution. Specifically, we consider churn in the asynchronous message passing system model. We prove that there does not exist an algorithm that can handle unlimited adversarial churn. We then focus on cooperative unlimited churn. We define the Infinite Unlimited Churn Problem by specifying the properties of a churn handing algorithm. We distinguish fair and unfair types of the problem. We prove that there is no local solution to the Fair Infinite Unlimited Churn Problem. We then present an algorithm that solves the unfair version of the problem while maintaining a linear topology, i.e. topological sort. We extend our algorithm to handle skip lists and skip graphs.

To the best of our knowledge, this paper is the first systematic study of unlimited infinite churn.

Related work. Independently of peer-to-peer overlay networks, several papers [24, 27, 36] address determination of the rate of churn, which is a difficult task itself. Fundamental problems in Distributed Computing, such as Agreement, were also studied in the context of churn [5, 6, 20], however their inherently global setting makes them unsuitable for building peer-to-peer overlays.

Usually, peer-to-peer overlay networks are designed to have redundant links so that they can withstand limited peer departure (that is, limited adversarial churn in our terminology) [7–9, 18]. Many papers address repairing the topology after determining a process

unexpectedly left the overlay network [1, 6, 15, 21, 32, 33]. Limited churn also enables the possibility to maintain overlay services while adjusting [4, 26].

Another approach deals with self-stabilizing peer-to-peer overlay maintenance algorithms [11, 12, 14, 16, 17, 22, 23, 25, 29, 30, 34] that enable the peer-to-peer overlay network to recover from an arbitrary topology disruption, once it stops. That is, self-stabilizing algorithms may handle unlimited but finite adversarial churn. In these algorithms, churn is handled implicitly: topology changes are assumed to eventually stop, after this the system is designed to recover. To handle the possibility of initial incorrect state, the use of oracles (that is, abstract entities that provide information about the overlay network) may be necessary [16, 30]. The recent trend is to provide a general framework that can be instantiated for various overlay networks [11, 25].

Overall, previous studies in the context of peer-to-peer overlay networks consider *limited* and/or *finite* churn, while this paper focuses on unlimited and infinite churn.

2 Model and Problem Statement

Peer-to-peer overlay networks, topology. A peer-to-peer overlay network consists of a set of processes with unique identifiers. We refer to processes and their identifiers interchangeably. Processes communicate by message passing. A process stores identifiers of other processes in its memory. Process a is a *neighbor* of process b if b stores the identifier of a. Note that b is not necessarily a neighbor of a. A process may send a message to any of its neighbors. Message routing from the sender to the receiver is carried out by the underlying network. A process may send a message only to the receiver with a specific id, i.e. we do not consider broadcasts or multicasts. Communication channels are FIFO with unlimited message capacity.

A *structured* peer-to-peer overlay network maintains a particular topology. One of the basic topologies is *linear*, or a topological sort, where each process b has two neighbors $a < b$ and $b > c$ such that a is the highest id in the overlay network that less than b and c is the lowest id greater than b. Consider a particular topology. A *cut-set* is a (proper) subset of processes of the network such that the removal of these processes and their incident edges disconnects the network. It is known that if a network topology is not a complete graph, it has a cut-set. Since a peer-to-peer overlay network maintains its connectivity by storing identifiers in the memory of other processes, once disconnected it may not re-connect. Hence, a peer-to-peer overlay network must not become disconnected either through the actions of the algorithm or through churn actions.

Searching, joining and leaving the overlay network. The primary use of a peer-to-peer overlay network is to determine whether a certain identifier is present in the overlay network. A search request message bearing an identifier, may appear in the incoming channel of any process that has already joined the overlay network. The request is routed until either the identifier is found or its absence is determined.

A process may request to join the overlay network. We abstract bootstrapping by assuming that a join request, bearing the joining process identifier, appears in an incoming

channel of any process that has already joined the overlay network. A process that joined the overlay network may leave it in two ways. In *adversarial churn* a leaving process just exits the overlay network without participating in further algorithm actions. In *cooperative churn* a leaving process sends a request to leave the overlay network. The leaving process exits only after it is allowed to do so by the algorithm. A process may join the overlay network and then leave. However, a process that left the overlay network may not join it again with the same identifier. A join or leave request is a *churn request* and the corresponding join or leave message is a *churn message*. When a leaving process exits the overlay network, the messages in its incoming channels are lost. However, the messages sent from this process before exiting remain in the incoming channel of the receiving process.

Infinite and unlimited churn definitions. Churn is *infinite* if the number of churn requests in a computation may not be bounded by a constant either known or unknown to the algorithm. Churn is *unlimited* if the number of concurrent churn requests in the overlay network is not bounded by a constant either known or unknown to the algorithm. Note that unlimited churn means that potentially all processes that are presently in the overlay network may request to leave. Note also that the two properties are orthogonal. For example, churn may be finite but unlimited: all processes may request to leave but no more join or leave requests are forthcoming. Alternatively, in infinite limited churn, there may be a infinite total number of join or leave requests but only, for example, five of them in any given state.

In this paper we only consider infinite unlimited churn.

Churn algorithm. A churn algorithm handles churn requests. For each process, an algorithm specifies a set of variables and actions. An *action* is of the form $\langle label \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$ where *label* differentiates actions, *guard* is a predicate over local variables, and *command* is a sequence of statements that are executed *atomically*. The execution of an action transitions the overlay network from one state to another. An algorithm *computation* is an infinite fair sequence of such states. We assume two kinds of fairness of computation: weak fairness of action execution and fair message receipt. *Weak fairness* of action execution means that if an action is enabled in all but finitely many states of the computation then this action is executed infinitely often. *Fair message receipt* means that if the computation contains a state where there is a message in a channel, this computation also contains a later state where this message is not present in the channel, i.e. the message is received. We place no bounds on message propagation delay or relative process execution speeds, i.e. we consider fully asynchronous computations.

Algorithm locality. A churn request may potentially be far, i.e. a large number of hops, from the place where the topology maintenance operation needs to occur. *Place of join* for a join request of process x, is the pair of processes y and z that already joined the overlay network, such that y has the greatest identifier less than x and z has the smallest identifier greater than x. In every particular state of the overlay network, for any join request, there is a unique place of join. Note that as the algorithm progresses and other processes join or

leave the overlay network, the place of join may change. *Place of leave* for a leave request of process x is defined similarly. *Place of churn* is a place of join or leave.

A network topology is *expansive* if there exists a constant m independent of the network size such that for every pair of processes x and y where the distance between x and y is greater than m, a finite number of processes may be added m hops away from x to increase the distance between x and y by at least one. This constant m is the *expansion vicinity* of the topology. In other words, in an expansive topology, every pair of processes far enough away may be further separated by adding more processes to the network without modifying the expansion vicinity of one member of the pair. Note that a completely connected topology is not expansive since the distance between any pair of processes is always one. However, a lot of practical peer-to-peer overlay network topologies are expansive. For example, a linear topology is expansive with expansion vicinity of 1 since the distance between any pair of processes at least two hops away may be increased by one if a process is added outside the neighborhood of one member of the pair.

A churn algorithm is *local* if there exists a constant l independent of the overlay network size, such that only processes within l hops from the place of churn need to take steps to satisfy this churn request. The minimum such constant l is the *locality* of the algorithm. Note that a local algorithm may maintain only an expansive topology, and that the expansive vicinity of this topology must be greater than the locality of the algorithm.

The Infinite Churn Problem. A *link* is the state of channels between a pair of neighbor processes. As a churn algorithm services requests, it may temporarily violate the overlay network topology that is being maintained. A *transitional link* violates the overlay network topology while a *stable link* conforms to it. An algorithm that solves the infinite churn problem conforms to a combination of the following properties:

request progress: if there is a churn request in the overlay network, some churn request is eventually satisfied;

fair request: if there is a churn request in the overlay network, this churn request is eventually satisfied;

terminating transition: every transitional link eventually becomes stable;

message progress: a message in a stable link is either delivered or forwarded closer to the destination;

message safety: a message in a transitional link is not lost.

Note that the fair request property implies the request progress property. The converse is not necessarily true.

Definition 1. The Unfair Infinite Unlimited Churn Problem *is the combination of request progress, terminating transition, message progress and message safety properties.*

Definition 2. The Fair Infinite Unlimited Churn Problem *is the combination of fair request, terminating transition, message progress and message safety properties.*

In other words, Fair Infinite Unlimited Churn guarantees that every churn request is eventually satisfied while Unfair Infinite Unlimited Churn does not.

3 Impossibilities

Adversarial churn.

Theorem 1. *There does not exist a solution for unlimited adversarial churn if the maintained topology is not fully connected.*

Proof: Assume there is an algorithm $\mathcal{A}$ that maintains a topology that is not fully connected such that $\mathcal{A}$ is resilient against unlimited adversarial churn. Let N be the number of processes in the overlay network in some global state. If the topology is not fully connected, there exists a cut-set P of processes whose cardinality is less than N. Since $\mathcal{A}$ handles unlimited churn, it should handle the departure of every process in P. However, since P is a cut-set, its departure disconnects the overlay network, which is not possible to handle. Hence, $\mathcal{A}$ is not able to handle unlimited adversarial churn. $\square$

Since adversarial unlimited churn cannot be handled, in the rest of the paper we are considering cooperative (unlimited) churn.

Fair local churn. Intuitively, the proof of the below theorem describes the continuous servicing of join requests before a distant churn request can reach its place of churn. This join chain keeps building in front of the distant churn request, precluding it from ever reaching the appropriate place.

Theorem 2. *There does not exist a local solution to the Fair Infinite Unlimited Churn Problem for an expansive overlay network topology.*

Proof: Assume there exists a local algorithm $\mathcal{A}$ with locality l that satisfies the Fair Infinite Unlimited Churn Problem while maintaining an expansive topology with expansion vicinity of $m > l$. Let us add a churn request with id x to the incoming channel of another process y whose distance d to the place of churn for x is greater than m and, therefore, l. Since the topology is expansive, there is a set of processes Z that are not part of the overlay network such that every process in this set can be added m hops away from y such that the distance d between y and the place of churn for x increases by at least one. Observe that $\mathcal{A}$ is assumed local with locality l that is less than expansion vicinity m. This means that every process in Z may be added to the overlay network without y having to take a step. Let us add the processes of Z to the overlay network. Let us then have y receive the churn request for x. Since $\mathcal{A}$ is a solution to the Fair Infinite Unlimited Churn Problem, y must forward this request closer to its place of churn. Suppose y forwards this request to some process u. However, since the distance from y to x's place of churn is at least $d+1$, the distance from u to this place of churn is at least d. We continue this procedure ad infinitum.

The resultant sequence is a computation of algorithm $\mathcal{A}$. Yet, there is a churn request of process x that is never satisfied. This means that $\mathcal{A}$ violates the fair request property of the Fair Infinite Unlimited Churn Problem, which contradicts our initial assumption. $\square$

Since fair local (infinite unlimited) churn is impossible, in the next section we address unfair local (infinite unlimited) churn.

4   Unfair Local Infinite Unlimited Churn for a Linear Topology

Linear topology under churn. In a linear topology, each process p maintains two identifiers: *left*, where it stores the largest identifier greater than p and *right*, where it stores the smallest identifier less than p. Processes are thus joined in a chain. For ease of exposition, we consider the chain laid out horizontally with higher-id processes to the right and lower-id processes to the left. The largest process stores positive infinity in its *right* variable; the smallest process stores negative infinity in *left*. A *left end* of a link is the smaller-id neighbor process. A *right end* is the greater-id process.

As a process joins or leaves the overlay network it may change the values of its own or its neighbors variables thus transitioning the link from one state to another. In a linear topology, a link is *transitional* if its left end is not a neighbor of its right end or vice versa. The link is *stable* otherwise. The largest and smallest processes may not leave. The links to the right of the largest process and to the left of the smallest processes are always stable. A process may leave the overlay network only after it has joined. We assume that in the initial state of the overlay network, all links are stable.

Algorithm description. We present a local algorithm *Unfair Infinite Unlimited Churn* ($\mathcal{UIUC}$) that satisfies the four properties of the Unfair Infinite Unlimited Churn Problem while maintaining a linear topology. The basic idea of the algorithm is to have the *handler* process with the smaller identifier coordinate churn requests to its immediate right. This handler considers one such request at a time. This serializes request processing and guarantees the accepted request's eventual completion.

The algorithm is shown in Figure 1. To maintain the topology, each process p has two variables: *left* and *right* with respective domains less than p and greater than p. Read-only variable *leaving* is set to **true** by the environment once the joined process wishes to leave the overlay network. Variable *busy* is used by the handler process to indicate whether it currently coordinates a churn request, or is initialized to **true** for a joining process. The incoming channel for process p is variable C.

The request is sent in the form of a single *join* or *leave* message. We assume that a *join* and, for symmetry, a *leave* message is inserted into an incoming channel of an arbitrary joined process in the overlay network.

Message *join* carries the identifier of the process wishing to join the overlay network. Message *leave* carries the id of the leaving process as well as the id of the process immediately to its right. Actions *joinRequest* and *leaveRequest* describe the processing of the two types of requests. If the receiver realizes that it is to the immediate right of the place of join or leave, and the receiver is not currently handling another request, i.e. *busy* $\neq$ **true**, and it does not want to leave, it starts handling the arrived request. Otherwise, the recipient process forwards the request to its left or right.

Request handling is illustrated in Figure 2. It is similar for join and leave and is divided into five stages. The first two stages are *setup* stages: they set up the channels for the links of the the joining process or for the processes that are the neighbors of the leaving process. The third and forth stages are *teardown stages*: they remove the channels of the links being

constant p // process identifier
variables

        $left, right$: ids of left and right neighbors, $\perp$ if undefined
        $leaving$: boolean, initially **false**, read only, application request
        $busy$: boolean, initially **false**; **true** when servicing a join/leave request or when joining
        C: incoming channel

actions
joinRequest: **join** $\in C \longrightarrow$

        receive join ($reqId$)
        if $(p < reqId < right)$ **and not** $leaving$ **and not** $busy$ **then**
            send sua($right$) **to** $reqId$
            $busy :=$ **true**
        else
            if $reqId < p$ **then**
                send join($reqId$) **to** $left$
            else
                send join($reqId$) **to** $right$

leaveRequest: **leave** $\in C \longrightarrow$

        receive leave($reqId, q$)
        if $reqId = right$ **and not** $leaving$ **and not** $busy$ **then**
            send sua($\perp$) **to** q
            $busy :=$ **true**
        else
            if $p <= reqId$ **then**
                send leave($reqId, q$) **to** $left$
            else
                send leave($reqId, q$) **to** $right$

setUpA: **sua** $\in C \longrightarrow$

        receive sua($reqId$) from q
        if $reqId \neq \perp$ **then** // Join 1.1 received
            $right := reqId$
            $left := q$
            send sua($\perp$) to $right$
        else // Join 1.2 or Leave 1 received
            $left := q$
            send sub to $left$

setUpB: **sub** $\in C \longrightarrow$

        receive sub from q
        if $q \neq right$ **then** // Join 2.2 or Leave 2 received
            send tda to $right$
            $right := q$
         else // Join 2.1 received
            send sub to $left$

tearDownA: **tda** $\in C \longrightarrow$

        receive tda from q
        if $q \neq left$ **then** // Join 3 or Leave 3.2 received
            send tdb to q
         else // Leave 3.1 received
            send tda to $right$

tearDownB: **tdb** $\in C \longrightarrow$

        receive tdb from q
        if $q \neq right$ **then** // Join 4 or Leave 4.2 received
            send ftd to q
            $busy :=$ **false**
        else // Leave 4.1 received
            send tdb to $left$

tranDone: **ftd** $\in C \longrightarrow$

        receive ftd from q
        if $leaving$ **then** // Leave 5 received, p may exit
            $right = \perp$
            $left = \perp$
        else
            $busy :=$ **false** // Join 5 received

Fig. 1. Algorithm $\mathcal{UIUC}$ for process p.

replaced. The last stage informs either the leaving process that it may exit, or the joining process that it may start coordinating its own churn requests. In the case of join, two links need to be set up, hence the setup stages are divided into two substages 1.1, 1.2, 2.1 and 2.2. Similarly, in the case of leave, the teardown stages are divided into substages because two links need to be torn down. The messages transmitted during corresponding stages are 1. set up A **sua**, 2. set up B **sub**, 3. tear down A **tda**, 4. tear down B **tdb** and 5. finish teardown **ftd**.
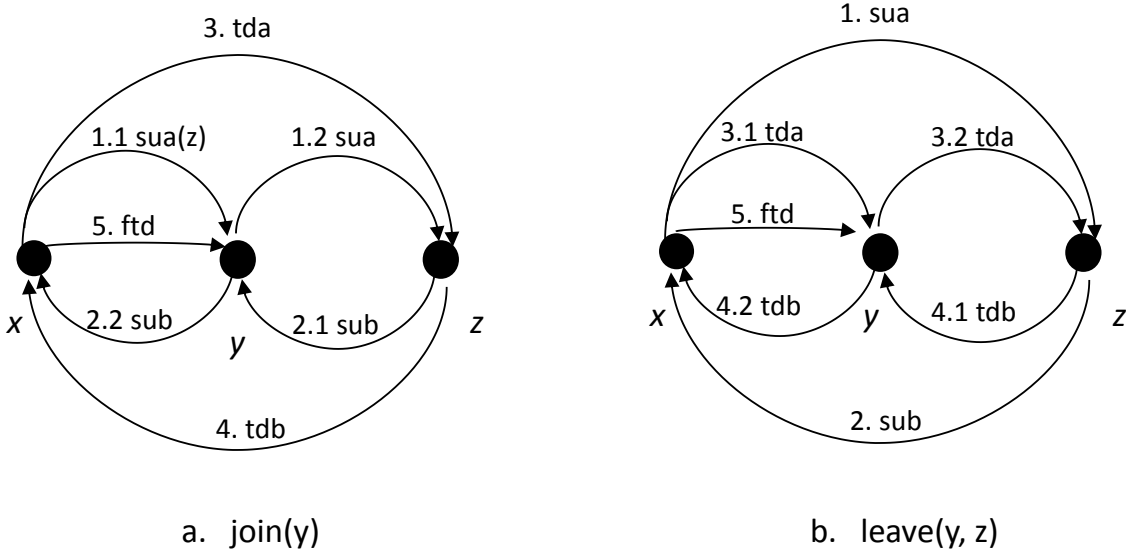

Fig. 2. $\mathcal{UIUC}$ join and leave request handling.

$\mathcal{UIUC}$ **correctness proof.** We denote message **tda** or **tdb** as **td***. Similarly, **su*** is **sua** or **sub**. Lemmas 1 and 2 follow immediately from the operation of the algorithm.

Lemma 1. *Message **td*** is the last message in the channel in every teardown stage. Message **su*** is the first message in a channel in every set up stage.*

Lemma 2. *The two links of a churning process transition through stages 1 through 5. No link participates in concurrent transitions.*

The below corollary follows from Lemma 2.

Corollary 1. *Every transitional link is eventually stable.*

Note that after the link is stable, it may transition again.

Lemma 3. *No message in a transitional link is lost.*

Proof: Observe that according to Lemma 1, in a teardown stage, the last message in the channel is a **td***. By Lemma 2, each stage, including the teardown stages, eventually completes. By the design of the algorithm, a teardown stage completes when a **td*** message is received. Since each channel is FIFO, this teardown message is received only after all other messages in the channel are received. That is, after the teardown stage completes, there are no messages in the channel that is torn down. According to Lemma 2, every channel in a transitional link is eventually torn down. Hence the lemma. □

Lemma 4. *Unless the link starts transitioning, a message in a stable link is either eventually delivered or forwarded to a process closer to destination.*

Proof: Consider a message in a channel of a stable link. We assume fair channel message receipt. This means, unless the channel is altered due to transitioning, the message is eventually processed by the recipient process. If this process is the destination, the message is delivered; if not, the recipient forwards it closer to the destination. □

Lemma 5. *If there is a churn request in the overlay network, some churn request is eventually satisfied.*

Proof: Consider state s_1 of the computation where there is a churn request message m in a channel of the overlay network. In this state, some other churn request processing may be under way. According to Lemma 1, the processing of all these requests eventually ends. Let s_2 be the state of the computation where all requests under way in s_1 are done. If a new request processing has started in s_2, then by applying Lemma 1 to it, we obtain the claim of this lemma.

Let us consider the case where no request processing has occurred between s_1 and s_2. According to Lemma 3, message m is sill in some channel of s_2. Since no requests are processed in s_2, there are no transitional links. That is, m is in a stable link. According to Lemma 4, m is forwarded towards its destination. Continuing this way, we observe that either m encounters a transitional link or arrives at a process p that handles this request. If m encounters a transitional link, this transition started after s_1 and the claim of the lemma follows. If m arrives at p, p may either be busy or not. If p is busy, it is handling a request that started after s_2. If p is not busy, it starts processing m. Hence the lemma. □

Only processes to the immediate right and left of the churning process are involved in the processing of the churn request. Hence the following lemma.

Lemma 6. *The locality of $\mathcal{UIUC}$ is 1.*

The following theorem follows from Corollary 1 and Lemmas 1,3,4,5 and 6.

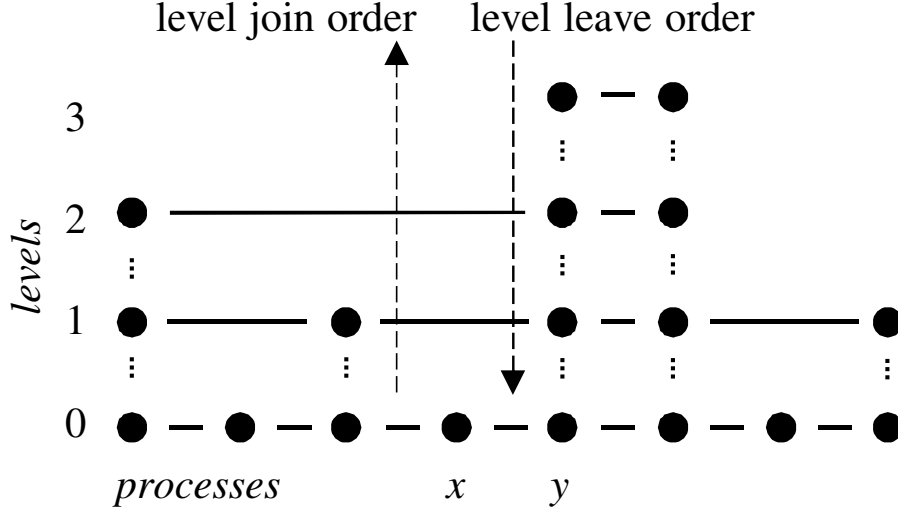**Theorem 3.** *Algorithm $\mathcal{UIUC}$ is local and it solves the Unfair Infinite Unlimited Churn Problem.*

Fig. 3. Processes joining and leaving an example skip-list.

5 $\mathcal{UIUC}$ Extensions to Skip List and Skip Graph and Further Work

Churn algorithm $\mathcal{UIUC}$ extends to more complicated topologies such as skip lists [29, 31] and skip graphs [2, 3, 13, 19]. In these topologies, the processes have links on multiple levels. The processes are linearized in the lowest level. In the higher levels, the processes have links to progressively more distant peers. These higher level links accelerate overlay network searches and other operations. See Figure 3 for an example skip list. To extend $\mathcal{UIUC}$ to such a structure a separate version of $\mathcal{UIUC}$ should be run at each level. The churn request should bear the level number to differentiate which level $\mathcal{UIUC}$ they belong to. The churning process should proceed up and down the levels as follows. A joining process first joins the first, linear, level, then the next and so on until it joins all the levels appropriate to the particular structure. The leaving process should proceed in reverse: the leaving process requests to leave the levels in decreasing order. For example, in Figure 3, a process y needs to first leave from level 3, then 2 and so on. While a joining process x needs to first join the overlay network at linearized level 0, then proceed to join level 1 and so on until it reaches the level appropriate for the particular structure.

As further research it is interesting to consider extensions of $\mathcal{UIUC}$ to ring structures such as Chord [35] or Hyperring [7]. Another important area of inquiry is addition of limited adversarial churn. This problem is difficult to address in the asynchronous message passing model where the exited process may not be differentiated from a slow one. Oracles determining a process exit [16] may have to be used.

References

1. David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *SOSP*, pages 131–145, New York, NY, USA, 2001. ACM.

2. James Aspnes, Jonathan Kirsch, and Arvind Krishnamurthy. Load balancing and locality in range-queriable data structures. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 115–124. ACM, 2004.

3. James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37, 2007.

4. John Augustine, Anisur Rahaman Molla, Ehab Morsy, Gopal Pandurangan, Peter Robinson, and Eli Upfal. Storage and search in dynamic peer-to-peer networks. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 53–62. ACM, 2013.

5. John Augustine, Gopal Pandurangan, and Peter Robinson. Fast byzantine agreement in dynamic networks. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 74–83. ACM, 2013.

6. John Augustine, Gopal Pandurangan, Peter Robinson, Scott Roche, and Eli Upfal. Enabling robust and efficient distributed computation in dynamic peer-to-peer networks. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 350–369. IEEE, 2015.

7. Baruch Awerbuch and Christian Scheideler. The hyperring: a low-congestion deterministic data structure for distributed environments. In *SODA*, pages 318–327, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

8. Baruch Awerbuch and Christian Scheideler. Towards scalable and robust overlay networks. In *IPTPS*, 2007.

9. Baruch Awerbuch and Christian Scheideler. Towards a scalable and robust dht. *Theory of Computing Systems*, 45(2):234–260, 2009.

10. Salman A Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *arXiv preprint cs/0412017*, 2004.

11. Andrew Berns, Sukumar Ghosh, and Sriram V Pemmaraju. Building self-stabilizing overlay networks with the transitive closure framework. In *Symposium on Self-Stabilizing Systems*, pages 62–76. Springer, 2011.

12. Eddy Caron, Frédéric Desprez, Franck Petit, and Cédric Tedeschi. Snap-stabilizing prefix tree for peer-to-peer systems. *Parallel Processing Letters*, 20(1):15–30, 2010.

13. Thomas Clouser, Mikhail Nesterenko, and Christian Scheideler. Tiara: A self-stabilizing deterministic skip list and skip graph. *Theor. Comput. Sci.*, 428:18–35, 2012.

14. Shlomi Dolev and Ronen I. Kat. Hypertree for self-stabilizing peer-to-peer systems. In *NCA*, pages 25–32, 2004.

15. Maximilian Drees, Robert Gmyr, and Christian Scheideler. Churn-and-dos-resistant overlay networks based on network reconfiguration. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, California USA, July 11-13, 2016. Proceedings*, pages 417–427. ACM, 2016.

16. Dianne Foreback, Andreas Koutsopoulos, Mikhail Nesterenko, Christian Scheideler, and Thim Strothmann. On stabilizing departures in overlay networks. In *Symposium on Self-Stabilizing Systems*, pages 48–62. Springer, 2014.

17. Dominik Gall, Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. Time complexity of distributed topological self-stabilization: The case of graph linearization. In *LATIN*, pages 294–305, 2010.

18. Sébastien Gambs, Rachid Guerraoui, Hamza Harkous, Florian Huc, and Anne-Marie Kermarrec. Scalable and secure polling in dynamic distributed networks. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 181–190. IEEE, 2012.

19. Michael T Goodrich, Michael J Nelson, and Jonathan Z Sun. The rainbow skip graph: a fault-tolerant constant-degree distributed data structure. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 384–393. Society for Industrial and Applied Mathematics, 2006.

20. Rachid Guerraoui, Florian Huc, and Anne-Marie Kermarrec. Highly dynamic distributed computing with byzantine failures. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 176–183. ACM, 2013.

21. Thomas P Hayes, Jared Saia, and Amitabh Trehan. The forgiving graph: a distributed data structure for low stretch under adversarial attack. *Distributed Computing*, 25(4):261–278, 2012.

22. R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *PODC*, pages 131–140, 2009.

23. Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. Skip+: A self-stabilizing skip graph. *Journal of the ACM (JACM)*, 61(6):36, 2014.

24. Steven Y Ko, Imranul Hoque, and Indranil Gupta. Using tractable and realistic churn models to analyze quiescence behavior of distributed protocols. In *Reliable Distributed Systems, 2008. SRDS'08. IEEE Symposium on*, pages 259–268, 2008.

25. Andreas Koutsopoulos, Christian Scheideler, and Thim Strothmann. Towards a universal approach for the finite departure problem in overlay networks. In Andrzej Pelc and Alexander A. Schwarzmann, editors, *Stabilization,*

Safety, and Security of Distributed Systems - 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings, volume 9212 of *Lecture Notes in Computer Science*, pages 201–216. Springer, 2015.

26. Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. Towards worst-case churn resistant peer-to-peer systems. *Distributed Computing*, 22(4):249–267, 2010.

27. Giuliano Mega, Alberto Montresor, and Gian Pietro Picco. On churn and communication delays in social overlays. In *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*, pages 214–224. IEEE, 2012.

28. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1(2012):28, 2008.

29. Rizal Mohd Nor, Mikhail Nesterenko, and Christian Scheideler. Corona: A stabilizing deterministic message-passing skip list. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 356–370, Grenoble, France, October 2011.

30. Rizal Mohd Nor, Mikhail Nesterenko, and Sébastien Tixeuil. Linearizing peer-to-peer systems with oracles. In *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, pages 221–236, 2013.

31. William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

32. Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, London, UK, 2001. Springer-Verlag.

33. Jared Saia and Amitabh Trehan. Picking up the pieces: Self-healing in reconfigurable networks. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

34. Ayman Shaker and Douglas S Reeves. Self-stabilizing structured ring topology p2p systems. In *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, pages 39–46. IEEE, 2005.

35. Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

36. Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *IMC*, pages 189–202, October 2006.